Linguaggi Funzionali

Sommario: 06-10 Maggio, 2021

- Linguaggi Funzionali: Principi e Proprietà
  - > Modello di Calcolo
  - > Trasparenza Referenziale
- Strutture Fondamentali:
  - > Espressione, Applicazione, Strategia di Valutazione
  - > Sistema di Tipi e Polimorfismo Generico
  - > API-ADT: Modulo Segnature, Moduli Implementazione
- Higher Order:
  - > Principio di Astrazione e Riuso di Codice
  - > Funzioni Higher Order: List.fold_left
  - > List.fold_right
  - > List.map
  - > List.filter
  - > Definizioni Tail Recursive

Linguaggi Funzionali: Principi e Proprietà

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
  - Senza Stato [1]: No Valori Modificabili
  - Trasparenza Referenziale:[2]

Definition

Sia $\mathtt{P}$ un programma ed $\mathtt{E_i}$ un suo termine (espressione) all'occorrenza $\mathtt{i}$. Se la computazione di $\mathtt{P}$ riduce $\mathtt{E_i}$ al valore $\mathtt{V_{E_i}}$, allora il programma $\mathtt{P}[\mathtt{E_i} \leftarrow \mathtt{V_{E_i}}]$ è equivalente a $\mathtt{P}$ e può rimpiazzarlo in ogni computazione.

Ad esempio. `f(x)+f(x)` $\equiv$ 2 * f(x), con + somma, e * prodotto.

- Correttezza e Verifica: Trasparenza Referenziale permette tecniche di prova piuttosto semplici.

[1] inteso come Memoria Modificabile

[2] $\mathtt{P}[\mathtt{E} \leftarrow \mathtt{V_E}]$ indica il rimpiazzamento di ogni occorrenza di $\mathtt{E}$ in $\mathtt{P}$ con $\mathtt{V_E}$

Programmazione Funzionale: Principi e Proprietà /2

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
  - Senza Stato
  - Trasparenza Referenziale
  - Correttezza e Verifica
  - Funzioni sono valori:
    - calcolabili da un'applicazione di funzione
    - passabili come argomenti a un'applicazione di funzione
  - Programmazione (astratta e) Higher Order
    - Astrazioni sui dati: Modellare valori con funzioni [3]
    - Astrazioni sul controllo: Modellare strutture di controllo con funzioni [4]
  - Computazione per Riduzione: Come per il Lambda-Calcolo (vedi Esercizio su SOS per Lambda-Calcolo)

[3] Es: Interi di Barendregt, `[n+1]`=λ`x y.x([n]x y)`, `env` e `store` in Lab.LPL

[4] Es: iteratori fold_left e fold_right in OCaml

Strutture Fondamentali: Funzioni, Applicazioni, Espressioni

- **Programmi** sono:
  - Una struttura (di Moduli) con definizioni di Tipi, Funzioni e Dati, che forma un ambiente (di bindings)
  - Una espressione da valutare in tale ambiente
- **Funzioni**:
  - Sono definite da espressioni:
    Ad es.: (OCaml) `fun x` $\rightarrow$ `let...in exp`
  - Possono contenere blocchi-espressione, `let_in_`
  - Sono Programming Units: Il corpo ha la stessa struttura del programma:
- **Applicazione**: Il Meccanismo di calcolo fondamentale
- **Trasmissione e Strategia di Valutazione**:
  - OCaml: Trasmissione per valore
  - Haskell: Valutazione esterna e Lazy costruttori

Strutture Fondamentali: Tipi

- Sistema dei Tipi (Pure OCaml).
  - **Tipi Basici Scalari**: int, float, bool, char, e (char)string
  - **T. Polimorfi e Variabili di tipo**. Ad es.: `'a`
  - **Tipi Basici Strutturati**: Tuples, List polimorfe (generiche).
    Ad es.: `('a * int) list`
  - **Tipi funzione**: $\rightarrow$.
    Ad es.: `'a list`$\rightarrow$`('a`$\rightarrow$`'a`$\rightarrow$`bool)`$\rightarrow$`'a list`
  - **Tipi Concreti**: Record e Variant Types.
    - **Definiti** mediante costrutto:
      type typeName = typeExpression
    - **Record**. Ad es.:
      `type ratio = {num:int; denum:int}`
    - **Variant o Algebrici**. Ad es.:
      `type ('a,'b) myType = Either of 'a | Or of 'b;;`
    - **Variant Ricorsivi**. Ad es.:
      `type 'a myList = Nil | Cons of 'a * 'a myList;;`
  - **Tipi Astratti**

Tipi Algebrici o Concreti

Definition (Tipi di Dato Algebrico)

I Tipi di Dato Algebrico sono collezioni di valori (anche strutturati), detti "termini", definiti per iniezione a partire da un insieme finito $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ di funzioni iniettive, dette "costruttori", di nome $\mathtt{g_i}$, e segnatura $\mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}$ dati, dove $\mathtt{k} \geq 0$ e $\mathtt{T_G}$ sia il tipo dei valori definiti con $\mathtt{G}$. Allora l'insieme di tali termini è

$$\mathcal{D}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{D}^{\mathtt{T_{i_j}}}\},$$

dove abbiamo indicato con $\mathcal{D}^{\mathtt{T}}$ l'insieme dei valori di tipo $\mathtt{T}$.

Proposition (Presentazione e Identità Sintattica)

Presentazione. *Un valore algebrico, o termine, ha sempre forma* $\mathtt{g(v1, ..., vk)}$, *dove* $\mathtt{g}$ *sia un costruttore di arità* $\mathtt{k} \geq 0$ *e* $\mathtt{v1, \ldots, vk}$ *valori di tipo atteso dalla segnatura di* $\mathtt{g}$.
Identità Sintattica $\equiv$. *Siano* $\mathtt{g(v1, ..., vn)}$ *e* $\mathtt{g'(v1', ..., vm')}$ *due valori di uno stesso tipo di dato algebrico. Allora,*

$$\mathtt{g(v1, ..., vn)} = \mathtt{g'(v1', ..., vm')} \quad \textit{sse } \mathtt{g} \equiv \mathtt{g'} \wedge \mathtt{n} = \mathtt{m} \wedge \mathtt{v1} = \mathtt{v1'} \wedge ... \wedge \mathtt{vn} = \mathtt{vm'}$$

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

$$\begin{aligned} \mathtt{type\ T_G} = {} & \mathtt{g_1\ of\ (Texp_{1,1} \star ... \star Texp_{1,k_1})} \\ & \mid ... \\ & \mid \mathtt{g_n\ of\ (Texp_{n,1} \star ... \star Texpt_{n,k_n})} \end{aligned}$$

Dove: - ...

Ocaml: Tipi Algebrici o Concreti

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

$$\begin{aligned}\texttt{type } T_G = {} & g_1 \texttt{ of } (Texp_{1,1} \star \ldots \star Texp_{1,k_1}) \\ & \mid \ldots \\ & \mid g_n \texttt{ of } (Texp_{n,1} \star \ldots \star Texpt_{n,k_n})\end{aligned}$$

  - **Dove:**
    - T_G *è il nome del tipo e deve iniziare con un carattere* **minuscolo**
    - $g_1, .., g_n$ *sono i nomi dei costruttori e hanno primo carattere* **maiuscolo**
    - of $(Texp_{i,1} \star \ldots \star Texp_{i,k_i})$ *è omesso quando* $k_i = 0$*, i.e.* g_i *ha arità 0*
    - of $(Texp_{i,1} \star \ldots \star Texp_{i,k_i})$ *definisce la tupla dei tipi degli argomenti*
    - $Texp_{i,j}$ è una qualunque espressione di tipo Ocaml
    - l'ordine dei costruttori è inessenziale
    - Nel caso di tipi algebrici polimorfi:
      - T_G ha prefisso la lista $`a_1, \ldots, `a_r$ delle variabili generiche
      - $Texp_{i,j}$ può contenere variabili in $\{`a_1, \ldots, `a_r\}$ come libere

  - **Ricorsiva** T_G può occorrere come tipo in $Texp_{i,k_i}$

  - **Allocazione Dinamica** Ogni valutazione di $g_i(v_{i,1}, \ldots, v_{i,k_i})$ alloca dinamicamente una struttura di costruttore g_i avente k_i componenti di valore $v_{i,1}, \ldots, v_{i,k_i}$ in tale ordine.

  - **Structure Sharing** Condivisione struttura dei sotto-termini.

Programmare con Tipi Algebrici o Concreti

- Programmare con Tipi Algebrici richiede non solo saper:
  - **Definire tipi algebrici.**
    Esempio, alberi `('a,'b) gTree` in Ocaml
    `type('a,'b)gTree = E | L of 'a | R of ('b * ('a,'b) gTree list)`
  - **Esprimere valori** di un dato tipo algebrico.
    Esempio, l'albero `tree`:
    `let tree = R("root1", [L 1; L 2; R("root2", []); E])`

- anche saper:
  - **Visitare la Struttura**
    Esempio: l'albero `tree` è una foglia? Ha più di 1 sottoalbero?
  - **Accedere Componenti**.
    Esempio: se `tree` ha sottoalberi, qual'è il suo primo sottoalbero?
  - **Modificare Componenti**
    Esempio: Ocaml permette tipi algebrici a componenti modificabili

- Operazioni per Visita, Accesso e Modifica ottenuti:
  - Meccanismo di Pattern-Matching (Ocaml, e L funzionali)
  - Meccanismi ad hoc
  - Ricostruzione e Condivisione sottotermini

Pattern-Matching

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti.

- È realizzato mediante:
  - Patterns per dato tipo algebrico
  - Operazione Match per coppie pattern-termine

- Sia $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ l'insieme dei costruttori di un Tipo Algebrico $\mathtt{T_G}$, e indichiamo con $\mathtt{X}^{\mathtt{T_G}}$ un insieme delle variabili di tipo $\mathtt{T_G}$
  - **Termini**. $\mathcal{D}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{D}^{\mathtt{T_{i_j}}}\}$
    Ad esempio: `R("root1", [L 1; L 2; R("root2", []); E])`
    è un termine di `('a,'b)gTree`, in Ocaml.

  - **Patterns** $\mathcal{P}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{P}^{\mathtt{T_{i_j}}}\} \cup \mathtt{X}^{\mathtt{T_G}}$
    Ad esempio: `R(x, [L 1; L 2; y; z])`
    è un pattern con `x` variabile di tipo `string`, `y` e `z` di tipo `('a,'b)gTree`.
    - Tutte le variabili che occorrono in un pattern sono **variabili libere**
    - L'operazione match lega

- Operazione `Match` : $\mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow$ `Sostituzione`,
  - `Match(t,p)` definisce un ambiente i cui binding sono i legami tra le variabili (libere) del pattern `p` e i corrispondenti sotto-termini del termine `t`.

Pattern-Matching in Ocaml

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- Operazione Match: Dato $\mathtt{T_G}$,
  - $\mathtt{Match} : \mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow \mathtt{Sostituzione}$
  - Sostituzione è l'insieme dei legami di variabili-valori che rendono il pattern uguale al termine dato, oppure *fail* se tale insieme non esiste.
    Esempio: $\mathtt{Match(R("root1", [L\ 1; L\ 2; R("root2", []); E]), R(x, [L\ 1; L\ 2; y; z]))}$
    calcola la sostituzione $\{\mathtt{x = "root1", y = R("root2", []), z = E}\}$

- Ocaml incorpora l'operazione Match in un costrutto match_with.

- Costrutto match_with è un condizionale multi-way avente la seguente forma:

```
match term with
      | pattern_1 {where pred_1}^{0,1} -> exp_1
      | ...
      | pattern_n {where pred_n}^{0,1} -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e
$\mathtt{pattern}_1, \ldots, \mathtt{pattern}_n$ sono `n` patterns per lo stesso tipo `T` che devono fornire una copertura per `T` (ovvero, per ogni termine di `T` deve esistere un pattern nella copertura e una sostituzione per esso che rende il pattern identico al termine)
$\{\mathtt{where\ pred}_i\}^{0,1}$ predicato da soddisfare se presente.

Pattern-Matching in Ocaml: Esempi

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- Costrutto match_with è un condizionale multi-way avente la seguente forma:

```
match term with
      | pattern_1 {where pred_1}^{0,1} -> exp_1
      | ...
      | pattern_n {where pred_n}^{0,1} -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e ...
calcola $\texttt{exp}_i$ tale che `i` è il più piccolo indice per cui:

$$\texttt{Match(term, patter}_i\texttt{)} \neq \texttt{fail} \quad \texttt{OR (not pred}_i\texttt{)}$$

Esempio (Calcolo della sostituzione)

Definiamo un tipo per la presentazioni delle sostituzioni:

```
type 'a substs = Sub of 'a | Fail;;
```

ed usiamolo nel match di termini di tipo ('a, 'b)gTree *definito prima.*

```
let tree = R("root1", [L 1; L 2; R("root2", []); E]);;
match tree with R(x, [L 1; L 2; y; z]) -> Sub(x, y, z) | w -> Fail;;
```

Cosa otteniamo in Ocaml?

Esempio (Predicati e Selettori di gTree)

Definiamo i predicati isE, isL, isR per riconoscere i valori delle 3 diverse strutture :

```
let isE tree = match tree with E -> true | _ -> false;;
let isL tree = match tree with L _ -> true | _ -> false;;
let isR tree = match tree with R _ -> true | _ -> false;;
```

Pattern-Matching in Ocaml: Errori sulle Variabili Libere

Esempio (Errori sull'uso delle variabili nei pattern)

```
Definiamo in OCaml la seguente funzione:
          let equal x y =
              match x with
                | y -> true
                | _ -> false

Cosa definiamo in Ocaml? In particolare, si dica:
(a) Perche' REPL segnala: "warning 11: this match is unused ?
(b) Perche' REPL introduce il binding:
    val equal: 'a -> 'b -> bool = <fun>
assegnando ad equal il tipo sopra ?
(c) Cosa calcola REPL per l'espressione: equal 5 7 ?
```

Esempio (Abusi sull'uso del match)

```
Cosa dire delle seguenti 2 definizioni ? Ci piacciono ?
(a)
     let equal1 x = function
      |z when x = z -> true |_ -> false
(b)
     let equal2 x y = x = y
```

Tipi Astratti: API o Segnatura in OCaml

- Modulo Signature: ha un nome,
- Contiene la signature dei pubblici
- La signature è racchiusa tra **sig**...**end**
- Elenca i tipi esportati (e implementati dall'ADT)
- Elenca la segnatura di ogni operazione esportata
- La coerenza tra API e ADT è controllata dal sistema dei tipi.

```
module type RELAZIONE =
sig type ('a,'b) relazione
  val relazioneC: unit -> ('a,'b) relazione
  val addPair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val removePair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione
  val isIn: ('a,'b) relazione -> 'a -> 'b -> bool
  val getUno: ('a,'b) relazione -> 'b -> 'a list
  val getDue: ('a,'b) relazione -> 'a -> 'b list
end;;
```

Tipi Astratti: ADT e Implementazione

- Modulo ADT: Ne possono esistere più d'uno.
- Ogni ADT ha un nome,
- Contiene le implementazioni
- L'implementazione è racchiusa tra **struct**...**end:NomeSign.**
- Implementazione tipi esportati
- Implementazione operazioni esportate

```
module Relazione =
(struct
   type ('a,'b) relazione = ('a * 'b) list
   let relazioneC = fun () -> []
   let addPair r x y = (x,y)::r
   let removePair r x y =
       let g = fun u -> if u!=(x,y) then [u] else [] in
       fold_right(append)(map g r)[]
   let isIn r x y =
       fold_right (||) (map ((=)(x,y)) r) false
   let getUno r y =
       let g = fun u ->if(snd(u)=y)then[fst(u)]else[] in
       fold_right(append)(map g r) []
   let getDue r x =
       let g = fun u ->if(fst(u)=x)then[snd(u)]else[] in
       fold_right (append) (map g r) []
end:RELAZIONE);;
```

Tipi Astratti: Quali Additionals?

- **Anatomia di ADT OCaml**
  Nessuna differenza con quella di ADT Java

  - **Stato Concreto c:** Struttura Implementazione dei Va [5]
    - `type ('a,'b) relazione = ('a*'b) list`
    - In generale, una tupla (o record) di tipi, o algebrici
      type tree = {label: labelType; sons: tree list}
      type tree = Tree of labelType * tree list
  - **AF e I:** Funzione di Astrazione e Invariante
    - $\texttt{AF(c)} = \{(\texttt{x},\texttt{y}) \mid (\exists \texttt{n} \in [\texttt{0..length(c)} - \texttt{1}])\ \texttt{hd(tl}^{\texttt{n}}\ \texttt{c)} == (\texttt{x},\texttt{y})\}$ [6]
  - **Additionals:** Ma per valori senza stato
    - toString: Stringa di presentazione del Va
      $\texttt{toString} : (\texttt{'a,'b})\texttt{relazione} \rightarrow \texttt{String}$
    - elements: Lista degli elementi contenuti nel Va
      $\texttt{elements}(\texttt{'a,'b})\texttt{relazione} \rightarrow (\texttt{'a} * \texttt{'b})\texttt{list}$

[5] Va indica un valore astratto
[6] $\texttt{g}^{\texttt{n}}$ indica n composizioni di g

Riuso e Principio di Astrazione

- Una prima definizione:

```
let rec addAll acc = function
  |[ ] -> acc
  |x::xs -> addAll (acc+x) xs;;
```

- Una seconda definizione:

```
let rec rev acc = function
  |[ ] -> acc
  |x::xs -> rev (x::acc) xs;;
```

- Hanno identica forma (pattern)

Definition (Principio di Astrazione)

Evitare programmi che richiedono di dichiarare più volte una "stesso pattern di controllo". Le similarità devono essere individuate e astratte creando opportune funzioni che le implementino.

List.fold_left e Funzioni Higher Order

- Un'unica dichiarazione:

```
let rec fold_left g acc = function
  |[ ] -> acc
  |x::xs -> fold_left g (g acc x) xs;;
```

- Tanti usi diversi:

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
#addAll0 [1;2;3;4;5];;
- :  int = 15
#revT [1;2;3;4;5];;
- :  int list = [5; 4; 3; 2; 1]
```

Funzioni Higher Order

- Una funzione Higher Order ha funzioni come parametro: `g`

```
let rec fold_left g acc = function
  |[ ] -> acc
  |x::xs -> fold_left g (g acc x) xs;;
```

- oppure, calcola una funzione: `addAll,rev,addAll0,revT`

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
```

- oppure, entrambe le cose:

```
fold_left g
```

List.map

- Ancora pattern simili:

```
let rec squareEach = function
  |[ ] -> [ ]
  |x::xs -> (x * x)::(squareEach xs);;
let rec show = function
  |[ ] -> [ ]
  |x::xs -> (x^", ")::(show xs);;
```

- Un'unica funzione:

```
let rec map g = function
  |[ ] -> [ ]
  |x::xs -> (g x)::(map g xs);;
```

- tanti usi diversi:

```
let squareEach = map (fun x -> x*x);;
let show = map (fun x -> x^", ");;
```

List.filter

- Ancora pattern simili:

```
let rec pos = function
  |[ ] -> [ ]
  |x::xs -> if x >= 0 then x::(pos xs) else pos xs;;
let rec even = function
  |[ ] -> [ ]
  |x::xs -> if x mod 2 == 0 then x::(even xs) else even xs;;
```

- Un'unica funzione:

```
let rec filter g = function
  |[ ] -> [ ]
  |x::xs -> if g x then x::(filter g xs) else filter g xs;;
```

- tanti usi diversi:

```
let pos = filter (fun x -> x >= 0);;
let even = filter (fun x -> x mod 2 == 0);;
```

List.fold_right

- Ancora pattern simili:

```
let rec sub fin = function
  |[ ] -> fin
  |x::xs -> x - (sub xs);;
let rec toString fin = function
  |[ ] -> fin
  |x::xs -> x^(toString xs);;
```

- Un'unica funzione:

```
let fold_right g xx fin = match xx with
  |[ ] -> fin
  |x::xs -> g x (forld_right g xs fin);;
```

- tanti usi diversi:

```
let sub fin = fun xx -> fold_rigth (-) xx fin;;
let toString fin = fun xx -> fold_right (^) xx fin;;
```

Riuso, Efficienza, Definizioni Tail Recursive

- **Astrazione di Pattern**. Invece di tante dichiarazioni `g1,...,gn` che instanziano in modo differente uno stesso pattern, nel programma si fa uso di una sola funzione `G` higher order, avente pattern che astrae ciascuna di tali istanze e si rimpiazza ogni `gi` con un'instanza di `G`, diversa e specifica per `gi`.

- **Proprietà**. Proprietà provate sul pattern `G` possono essere opportunamente riformulate ed utilizzate in ogni sua instanza `gi`.
  - **Proviamo**. $\forall$ `(g:'a->'b->'b, xx:'a list, fin:'b)`
    `fold_right g xx fin == fold_left (eX g) fin (rev xx)`[7]
    dove: `eX h = fun a x -> h x a`

- **Efficienza**. Una ri-definizione più efficiente di `G` conduce a una corrispondente maggiore efficienza di ciascun instanza fornita per `g1,...,gn`, fornendo possibilmente pattern di calcolo alternativo e più efficiente per ognuna di tali funzioni `g1,...,gn`.
  - **Tail Recursive**. Mostriamo che le funzioni higher order, `fold_rigth`, `map`, `filter` hanno (un pattern tail recursive e quindi) una definizione tail recursive (in slide "folder_right è tail recursive" e successive).

[7] Una dimostrazione é nella slide successiva e rev indica la funzione che calcola la reverse di lista (definita anche in slide "Riuso e Principio di Astrazione")

Possiamo esprimere fold_right con fold_left

- **Lemma1** ∀ ((g:'a->'b->'b),(z:'a),(fin:b))
  g z fin == fold_left(eX g) fin [z]
- **Lemma2** ∀ (g:'a->'b->'b, xx,yy:'a list, fin:'b)
  fold_left g fin (xx@yy) == fold_left g (fold_left g fin xx) yy
- **Lemma3** ∀ ((x:'a),(xx:'a list)) [x]@xx == x::xx
- **Lemma4** ∀ (x:'a), rev[x] == [x] [8]
- **Lemma5** ∀ (xx:'a list, yy:'a list),
  rev(xx@yy) == (rev yy)@(rev xx)

- **Proviamo**. ∀ (g:'a->'b->'b, xx:'a list, fin:'b)
  fold_right g xx fin == fold_left (eX g) fin (rev xx)
  dove: eX h = fun a x -> h x a
- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:
  ∀ (g:'a->'b->'b, zz:'a list, fin:'b) && List.length zz < k,
  fold_right g xx fin == fold_left (eX g) fin (rev xx)
  - Sia xx == z::zz.

[8] rev indica la funzione che calcola la reverse di lista (definita anche in slide "Riuso e Principio di Astrazione")

La dimostrazione

- **L1.** `g z fin == fold_left(eX g) fin [z]`
- **L2.** `fold_left g fin (xx@yy) == fold_left g(fold_left g fin xx)yy`
- **L3.** `[x]@xx == x::xx`
- **L4.** `rev[x] == [x]`
- **L5.** `rev(xx@yy) == (rev yy)@(rev xx)`
- **Proviamo**. ∀ `(g:'a->'b->'b, xx:'a list, fin:'b)`
  `fold_right g xx fin == fold_left (eX g) fin (rev xx)`
  dove: `eX h = fun a x -> h x a`
- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:
  ∀ `(g:'a->'b->'b, zz:'a list, fin:'b) && List.length zz < k`,
  `fold_right g xx fin == fold_left(eX g) fin (rev xx)`
  - Sia `xx == z::zz`.
  - `fold_right g (z::zz) fin`
  - `== g z (fold_right g zz fin)` — `fold_right`
  - `== g z (fold_left(eX g)fin(rev zz))` — ind. ass.
  - `== fold_left(eX g)(fold_left(eX g)fin(rev zz))[z]` — Lemma1
  - `== fold_left(eX g)fin((rev zz)@[z])` — Lemma2
  - `== fold_left(eX g)fin((rev zz)@(rev[z]))` — Lemma4
  - `== fold_left(eX g)fin(rev([z]@zz))` — Lemma5
  - `== fold_left(eX g)fin(rev(z::zz))` — Lemma3

Efficienza: Definizioni Tail Recursive

- **fold_right**

  ```
  fold_right = fun g xx fin -> fold_left(eX g) fin (rev xx)
  ```

  Usate: `fold_left`, `rev` hanno definizioni Tail Recursive.

- **map**

  ```
  map = fun g xx -> fold_right (fun x a -> (g x)::a) xx [ ]
  ```

  Usate: `fold_rigth` ha definizione Tail Recursive.

- **filter**

  ```
  filter = fun p xx ->
           let g = fun x a -> if (p x) then x::a else a in
           fold_right g xx [ ]
  ```

  Usate: `fold_right` ha definizione Tail Recursive.

Esercizi

- Perchè nella programmazione Funzionale `List.fold_left` è considerato un iteratore? Si spieghi cosa è valutato ad ogni iterazione e come variano gli eventuali indici dell'iterazione.
- (a) Cosa calcola `map (+5) (map (*2) xx)`?
  (b) Cosa si può dire in generale, su un pattern della forma: `map f (map g xx)`?
- (a) Si dia una definizione di `length` che calcola la lunghezza di una lista
  (b) Sapreste dare una definizione Tail Recursive di `length`
- Sia indList una funzione che applicata ad una arbitraria lista, calcola la lista di coppie avente come primo componente la posizione della coppia nella lista e come secondo componente il valore in quella posizione. Ad esempio indList [56;12;-7] deve calcolare [(1,56),(2,12),(3,-7)].
  (a) Si dia una definizione di indList;
  (b) Si dia una definizione iterativa e Tail Recursive di indList che utilizzi le sole operazioni: `fst, snd, List.fold_left, List.length`.
- La funzione `zip` è un trasformatore, applicato ad una coppia di liste di stessa lunghezza calcola la lista contenente in ogni posizione, la coppia avente primo e secondo componente uguale al valore contenuto nella stessa posizione nella prima e nella seconda lista.
  (a) Si dia una definizione di zip.
  (b) Si dia una definizione tail-recursive
- Anche la funzione flatten è un trasfomatore: Data una lista di liste di valori 't, flatten genera la lista di valori 't formata dalla concatenazione delle liste di 't.
  Si sia una definizione tail recursive di flatten.
- La funzione duplex rimuove ogni valore duplicato presente in una lista.
  (a) Si dia una definizione ricorsiva di duplex
  (b) Si dia una definizione tail recursive di duplex
- Il numero di occorrenze di un valore 't in una lista 't list può essere calcolato con un pattern simile a quello utilizzato nella definizione di isIn di Queue.ml, ovvero:
  `occ x xx = List.length (filter ((=)x) xx)`.
  Possiamo però, fornire altre definizioni tail recursive che evitano l'uso di filter e length. Si mostri come.
- Le occorrenze di un valore 't in una lista 't list è una int list contenente le posizioni (numerate a partire da 1 per la prima) dele eventuali occorrenze del valore nella lista data.
  Possiamo fornire una funzione occList, definita tail recursive, che calcola la lista delle occorrenze di un valore in una lista?